```
/***************************************************************/
/*          File: comp_main.c                                 */
/*          Author: Hari Hampapuram                            */
/***************************************************************/
/* Associated files:
                    symboltab.c
                    moduleio.c
                    comp_misc.c
                    comp_scatter.c
                    sectiontab.c
                    comp_btarget.c
                    comp_bitstring.c
                    comp_reference.c
                    comp_src.c
                    comp_utils.c
                    error.c -- (only f_error is used, so we may be able to avoid this)


*/
/* Functions defined are:
                    1. void create_global_symbol_table(Module inmod,
                                          SymbolTable *sym_tab)
                    2.void update_global_symbol_table(SymbolTable *sym_tab,
                                          BtargetTable *btarget_table)
                    3.void update_local_symbol_table(SourceInfo ls,
                                          BtargetTable *btarget_table)
                    4. static void read_input_module(Module inmod);
                    5. static void allocate_space_for_output(Module inmod,
                                          Module out_mod)
                    6  main()
*/

/*****************************************************************/
/*
A very broad outline of the compressor code:
     STEP 1  Read the command and parse it.
     STEP 2. Read the object module into a data structure.
     STEP 3. Process the binary partition.
          STEP 3.1 Process text section.
               STEP 3.1.1 Collect branch targets.
               STEP 3.1.2 Collect source file boundaries, if any.
               STEP 3.1.3 Compress the text section.
               NOTE:
               i) Processing each instruction involves checking if it is
               a branch target or not.
               ii) When source file boundaries are available it is probably
               best to process the binary string file by file and update the
               information in the source file partition simultaneously.
               iii) The new addresses of branch targets are to be noted
               somewhere.
               iv) As the text gets compressed, the bitfield references in the
               reference table are to be updated.
               v) Finally, a pass through the reference table can update the
               values in the bitfields.
          STEP 3.2 Update the other sections in the binary partition.
     STEP 4. Update the rest of the object module and dump the object
     module.
*/
/***************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

/******* files from linker ***********/
```

```
#include "types.h"
#include "lifetypes.h"
#include "lifeobj.h"
#include "scatter_types.h"
#include "inum_map.h"
#include "salloc.h"
#include "stringtab.h"
#include "linktypes.h"
#include "libtypes.h"
#include "error.h"
#include "moduleio.h"
#include "libio.h"
#include "sectiontab.h"
#include "symboltab.h"
#include "symdump.h"
#include "symbolmap.h"
#include "sourcetab.h"
#include "mergeglobal.h"
#include "mergebinary.h"
#include "mergedebug.h"
#include "cmdline.h"

#include "stabs_updref.h"
/******* files from compressor *********/
#include "compressor.h"
#include "comp_bitstring.h"
#include "comp_btarget.h"
#include "comp_misc.h"
#include "comp_reference.h"
#include "comp_scatter.h"
#include "comp_src.h"
#include "comp_utils.h"
#include "dump_structs.h"

#include "cmd_parse.h"

AllGlobals G_all_globals;
static char *version_str = "tmcomp version 1.3 of " __DATE__ " " __TIME__ "\n";
/*****************************************************************/
/******  Some global variables to get compilation going!    *********/
/*****************************************************************/
SectionId no_sections;/*used in sct_add(), and write_reloc_map() in
                sectiontab.c, none of which is called by any compressor
                routines and (I think!) none of the routines called by
                the compressor routines. */
Section scttab[SectionId_max];/*used in sct_merge(), sct_add(), and
                write_reloc_map() in sectiontab.c, none of which is called
                by any compressor routines and (I think!) none of the
                routines called by the compressor routines. */
StringTable strtab;/** This is more of a tricky business. Several
                error routines do use the string table  and the initialization
                and usage is not clear. Hopefully, I can mimic lifelink.c
                and lifedump.c. -- after some study, I find that
                lifedump.c has a dummy declaration and from the linker's
                code, only some strings like machine_str, version, some
                section_names are added into the string table by st_add
                (or via StringId_map). So I will just have this dummy
                declaration and proceeed. **/
SourceTable srctab; /* used only in srctab_merge() which I dont use
                directly or indirectly. So this will be just a dummy
                declarator. */
Module outmod; /** used in srctab_merge() - so ok as a dummy
                declaration. */
unsigned long begin_memory; /** used is sectiontab.c for
                loading. Thus this is not needed for the compressor. */
boolean supress_warnings;
```

```
/*****************************************************************/
/******                          Global Symbol Table
***********/
/*****************************************************************/

void
create_global_symbol_table(Module inmod, SymbolTable *sym_tab)
{
byte *mem_addr = inmod -> global_image + inmod -> symbol_tbl_offs;
unsigned long  num_symbols = inmod -> symbol_tbl_size;
Symbol new_symbol;

        sym_initcreate(128);
        if (symtab_init(sym_tab, num_symbols) == 0){
                STD_ERROR("Symbol table initialization failure; out of memory?");
        }

        while ( num_symbols-- != 0){
                if (( new_symbol = sym_create()) == NULL){
                        STD_ERROR("Could not create new symbol." );
                }
                mem_addr = unpackSymbolDescr(new_symbol,mem_addr);
                new_symbol -> defmod = inmod;
                symtab_add(sym_tab,new_symbol);
        }
}

/*This should be called before updating the local symbol table.*/
/*This consults the local symbol table to get the address of a symbol
in the uncompressed object module. */
void
update_debug_section(Module md, SourceInfo ls,
                     SymbolTable *global_symtab, InumMap *inum_map)
{
ReftabEntry ref_ptr;
unsigned char * btarget_name;
unsigned char *debug_section;
Symbol global_symbol;
SymDescr sym;
StringId btarget_stringid;
byte *lptr;
StringId cnt;
unsigned long first_offset_uncompressed, second_offset_uncompressed;
unsigned long compressed_size;
unsigned char *current_string;
unsigned int current_offset;
char error_str[MAX_LENGTH_OF_ERROR];
int found;

debug_section = ls->module->source_image + ls->debug_info_offs;
initDebugRef(debug_section);

while ((ref_ptr = getDebugRef()) != NULL){
  btarget_name = ref_ptr->label;
  current_offset = 0;
  current_string = (unsigned char *)md->string_tbl;
  while (strcmp((const char *)btarget_name, (const char *)current_string)!=0){
    current_offset +=  strlen((const char *)current_string)+1;
    if (current_offset > md->string_tbl_size){
         sprintf(error_str, "string %s not found in string table\n", btarget_name);
      STD_ERROR(error_str);
    }
    current_string +=  strlen((const char *)current_string)+1;
  }
  btarget_stringid = current_offset;
```

```
  /*  search local symbol table for btarget_stringid; */
  /*  if found get the new value from the local symbol table;*/
  lptr = ls->module->source_image + ls->sym_tbl_offs;
        found = 0;
  for (cnt = ls->sym_tbl_size; cnt != 0; --cnt) {
    unpackSymbolDescr(&sym, lptr);
    if (sym.name == btarget_stringid) {
      first_offset_uncompressed = sym.value.lo;
          found = 1;
          break;
    }
      lptr += LIFE_Obj_sym_size;
  }

  /*  if not search global symbol table; */
  /*  if found get the new value from btarget table or inum map table;*/
  /* if not found report error;*/
        if (!found){
                if ((global_symbol = symtab_lookup(global_symtab, btarget_stringid)) ==
NULL){
                        STD_ERROR("Symbol not found in either local or global tables");
                }else{
                        first_offset_uncompressed = global_symbol -> value.lo;
                }
        }

  second_offset_uncompressed = first_offset_uncompressed + ref_ptr->size-1;

  compressed_size = inum_map_get_offset(inum_map, second_offset_uncompressed) -
    inum_map_get_offset(inum_map, first_offset_uncompressed) +
      inum_map_get_size(inum_map, second_offset_uncompressed);
  putDebugRef(ref_ptr, compressed_size);
}
finiDebugRef();
}

void
update_global_symbol_table(SymbolTable *sym_tab, BtargetTable *btarget_table)
{
Symbol symbol;
BtargetDescr *btarget;

        symtab_next(sym_tab, TRUE);
        while ((symbol = symtab_next(sym_tab, FALSE)) != NULL){
                if ( IS_TEXT_SECTION_ID(symbol -> section, symbol -> defmod) ){
                        if ((btarget =
                                btarget_table_lookup(btarget_table, &(symbol -> value)))
 == NULL){
                                STD_ERROR("Branch target lookup failure.");
                        }
                        symbol -> value = btarget -> new_address;
                };
        }
}

void
update_local_symbol_table(SourceInfo ls, BtargetTable *btarget_table)
{
SymDescr sym;
BtargetDescr *btarget;
StringId             cnt;
Section              lsct;   /* local section referenced by sym */
byte                 *lptr;

        lptr = ls->module->source_image + ls->sym_tbl_offs;
        for (cnt = ls->sym_tbl_size; cnt != 0; --cnt) {
        unpackSymbolDescr(&sym, lptr);
        if (IS_TEXT_SECTION_ID(sym.section, ls->module)) {
                        if ((btarget =
                                btarget_table_lookup(btarget_table, &(sym.value)))==NULL
){
                                STD_ERROR("WARNING:Symbol in local symbol table not in t
he btarget table.\n");
                        }else{
                                sym.value = btarget -> new_address;
                        }
                }
        lptr = packSymbolDescr(&sym, lptr);
    }

}

/*****************************************************************/
/*********            read input module             ************/
/*****************************************************************/
static void
read_input_module(Module inmod)
{
        md_readFile(inmod);
        sct_createAndLoad(inmod);
        if (/*src file partition exists*/ 1){
                src_initcreate(128);
                srctab_init(&(inmod -> source_table), 64);
                srctab_createAndLoad(inmod);
        }
        scatter_table_createAndLoad(inmod, &inmod->scatter_table);
        inmod -> string_tbl =
                (char *) (inmod -> global_image + inmod -> string_tbl_offs);
}

static void
allocate_space_for_output(Module inmod, Module out_mod)
{
unsigned long binary_size;

        binary_size = (inmod -> source_image - inmod -> binary_image);
        binary_size += ICACHE_BLOCK_SIZE; /* to make sure that
                there is enough place for padding. This may not be enough
                in all cases. */
        if ((out_mod -> binary_image = (byte *) malloc(binary_size)) == NULL){
                STD_ERROR("Memory allocation failure.");
        };
}

#if 0
void
pname_to_fname(char *pname, char *fname)
{
char *ptr;

        if ((ptr = strrchr(pname, '/')) == NULL){
                strcpy(fname, pname);
        } else {
                strcpy(fname, ptr+1);
        }
;

void basename(char *fname, char *bname)
{
char *ptr, *ptr2;
int i;
```

```
        if ((ptr = strrchr(fname, '.')) == NULL){
                strcpy(bname, fname);
        } else {
                for (ptr2 = fname, i=0; ptr2 != ptr; ptr2++, i++){
                        bname[i] = *ptr2;
                }
                bname[i] = '\0';
        }
}

#endif
void print_usage()
{
        fprintf(stderr, "Usage: comp <filename>");
}

/*****************************************************************/
/*************         main progarm              **************/
/*****************************************************************/
/*
        At  the top level, this function consists of the following steps.
        Details of each step can be found at the beginning of that
        step.
                Part I.  read input into data structures;
                Part II. process the text section;
                Part III. update the bitfields in the remaining sections;
                Part IV. Fill in the details for the global section;
                Part V. create memory image of header.
                Part VI. write output onto file
*/
void
main_compressor(char *inmod_name , char *outmod_base, char * mapfile_name)
{
FILE *output_file;
char error_str[MAX_LENGTH_OF_ERROR];
char outmod_name[MAX_MODNAME_LENGTH +1];
Module inmod, out_mod;
SymbolTable symbol_table;
Symbol symbol;
ReferenceTable ref_table;
Section text_section;
SectionId text_section_id, section_id;
BtargetTable btarget_table;
SourceFileIterator src_iterator;
SourceInfoDescr *s_info;
UncompressedBitstring uncomp_bitstring;
CompressedBitstring comp_bitstring;
int morefiles;
Address new_size;
Address first_file_base, tmp_address;
time_t          cal_time; /* current calendar time */
struct tm       *local_time;    /* current local time */
char                  *timestamp; /* current local time string*/
unsigned long    history_size; /* total linking history size (bytes) */
char                  cmdstring[MAX_CMD_LENGTH +1];
char                  history_string[MAX_HISTORY_LENGTH +1];
unsigned long current_file_offset, tmp_size;
unsigned long  size, binary_size;
byte * base_address, *section_start;
Section section;
byte *global_partition;
unsigned long global_partition_size;
byte *mptr;
int i;
unsigned long src_partition_size, scatter_table_size, tmp_value;
```

```
BtargetDescr *btarget;
InumMap *inum_map;

byte headbuf[128];
int pad_num; /*index variable for padding bytes at the end of
                              text section. */
int pad_size; /* number of padding bytes required at teh end of
                              the text section. */
int TESTING=1;/* setting this to 1 leaves the linking history
                              unchanged - convenient for testing. */
/* --------------------- ------------------------------------ */
/*              Part I.
                        i)  read input into data structures,
                        ii)      get memory for output binary image;
                        NOTE: Memory is obtained separately for the
                                    global image;
                               The header values are stored in the out_mod
                                    struct itself.
                               The source partition image is updated in the
                                 inmod and output from there directly.
                        iii)     create important data structures - global
                                 symbol table, reference table for text section,
                                 and the branch target table.             */
/* -------------------------------------------------------- */

#if 0
/* get inmod name and out_mod name */
        if (argc != 2) {
                print_usage();
                exit(1);
        }
        strcpy(inmod_name, argv[1]);
        pname_to_fname(inmod_name, fname);
        basename(fname, outmod_name);
        strcat(outmod_name, ".co");

#endif
/*      unlink(outmod_name);*/

        /* fprintf(stdout, "%s\n", version_str);*/
        sprintf(cmdstring,"tmcomp " );
        if (G_all_globals.flags.padoff){
                strcat(cmdstring, "-padoff ");
        }
        if (G_all_globals.flags.mapon){
                strcat(cmdstring, "-mapon ");
                if ((G_all_globals.mapfile = fopen(mapfile_name, "w")) == NULL){
                        sprintf(error_str, "Could not open map file: %s",
                                        mapfile_name);
                        STD_ERROR(error_str);
                }
        }
        if (G_all_globals.flags.shuffleoff){
                strcat(cmdstring, "-shuffleoff ");
        }
        strcat(cmdstring, inmod_name);
/*
        fprintf(stdout, "Effective command:%s\n", cmdstring);
*/
        if (TESTING) cmdstring[0]='\0';
        cmdstring[MAX_CMD_LENGTH] = '\0';

        salloc_init(1024, 256);

        inmod = md_create(inmod_name);
        read_input_module(inmod);
```

```
        if (inmod->flags & LIFE_Obj_is_compressed){
                        /* free all the data structures associated with the modules
                                and then free the module structure itself. */
                        free(inmod->memory_image);/* We have used md_readFile() and that

                                        allocates memory for the entire memory image. */

                        sct_destroy(inmod->section_tbl);
                        scatter_table_free(inmod->scatter_table);
                        srctab_destroy(&(inmod -> source_table));
                        free(inmod);
                        return;
        }
        if (inmod->flags & LIFE_Obj_is_exec){
                strncpy(outmod_name, outmod_base,
                        MAX_MODNAME_LENGTH - strlen(LIFECOMP_EXEC_EXTN));
                strcat(outmod_name, LIFECOMP_EXEC_EXTN);
        }else{
                strncpy(outmod_name, outmod_base,
                        MAX_MODNAME_LENGTH - strlen(LIFECOMP_EXTN));
                strcat(outmod_name, LIFECOMP_EXTN);
        }
        if (G_all_globals.flags.is_outmod_given){
                 strncpy(outmod_name, G_all_globals.out_mod_name,
                        MAX_MODNAME_LENGTH);
        }
        unlink(outmod_name);

        out_mod = md_create(outmod_name);
        out_mod -> scatter_table = scatter_table_create(64);
        /*COPY THE inmod scatter-table to out_mod. */
        scatter_table_cpy(out_mod -> scatter_table, inmod -> scatter_table);
        allocate_space_for_output(inmod, out_mod);
                                /*allocate space for the compressed
                                module, based on the sizes in inmod AND
                                initialize the pointers in out_mod.
                                Actually, space is allocated only
                                for binary partition. */

        if ((text_section = GET_SECTION(inmod, "text")) == NULL){
                STD_ERROR("No text section inthe input module.");
        };
        text_section_id = GET_SECTION_ID(inmod, "text");
        create_global_symbol_table(inmod, &symbol_table);
        create_reference_table(inmod, &ref_table, text_section);

        collect_branch_targets(&symbol_table, &btarget_table, inmod);
        /*dump_BtargetTable(&btarget_table);*/
        comp_bitstring.begin_address = out_mod -> binary_image;
        comp_bitstring.first_unused_address = out_mod -> binary_image;

        /* Create the inum_map table to be of size the number of
           instructions in the text section. */
        inum_map = inum_map_create(text_section -> size.lo);
/* -------------------------------------------------------- */
/*              Part II. process the text section
        For each source file,
                i)   get the corresponding part of
                the bitstring in teh binary partition,
                ii)  compress the bitstring,
                iii) update the link map entries to reflect the new
                boundaries of the text segment of that file.
                iv)  update the local symbol table entries of teh src file.

        After compressing the entire bitstring,
                i) update the global symbol table;
                ii)update the bitfields in the text segment;
                (to reflect the new values of the branch targets.)
                iii) pack the reference table in out_mod's binary image.

                NOTE:Compressing the bitstring also updates the reference
                table of the text section, to reflect the new positions
                and scatter types of the bitfields.
                        The compressed bitstring is built in out_mod's binary
                image.

*/
/* ------------------------------------------------------------ */
        if (/*src file partition exists*/ 1 ){
                begin_src_files_iterate(&src_iterator, &(inmod -> source_table));
                        /* start stepping through the
                                                        code file by file */
                first_file_base = src_iterator.current_link_map.base;
                reference_table_next(&ref_table, TRUE);
                /* The above initialization is needed for compress_bitstring.*/

                morefiles = 1;
                while (morefiles) {
                        if (IS_TEXT_SECTION_ID((src_iterator.current_link_map).section,
inmod)){
#if 0
                                tmp_address =
                                        Address_sub(src_iterator.current_link_map.base,

                                                first_file_base);
#endif
                                tmp_address = src_iterator.current_link_map.base;
                                Address_to_bit_vector(&tmp_address,
                                        sizeof(unsigned long)*8,
                                        (byte *)&current_file_offset);
                                uncomp_bitstring.first_ins_num = current_file_offset;
                        /*      printf("current_file_offset = %d\n", current_file_offset
);*/
                                current_file_offset *= INSTRUCTION_WIDTH_BYTES;
                                uncomp_bitstring.begin_address = inmod -> binary_image
                                        + text_section -> bitstring_offs + current_file_
offset;
                                uncomp_bitstring.current_address =
                                                uncomp_bitstring.begin_address;
                                Address_to_bit_vector(&(src_iterator.current_link_map.si
ze), sizeof(unsigned long)*8, (byte *)&tmp_size);
                        /*      printf("tmp_size= %d\n", tmp_size);*/
                        /* tmp_size is now the number of instructions. */
                                uncomp_bitstring.num_instructions = tmp_size;
                                compress_bitstring(&uncomp_bitstring, &comp_bitstring,
                                        &btarget_table, &ref_table, out_mod -> scatter_t
able, inum_map);
                                bit_vector_to_Address(&new_size,
                                                sizeof(unsigned long)*8,
                                                (byte *)&comp_bitstring.size_last_src_fi
le);
                                tmp_value = comp_bitstring.begin_last_src_file -
                                                        comp_bitstring.begin_address ;
                                bit_vector_to_Address(&tmp_address, sizeof(byte *)*8,
                                        (byte *)&tmp_value);
                                update_current_link_map(&src_iterator,
                                        &tmp_address, &new_size);
                                /*UPDATE LOCAL SYMBOL TABLE HERE */
                                s_info = get_current_src_descr(&src_iterator);

                                update_debug_section(inmod, s_info,
                                                        &symbol_table /*global table*/, inu
```

```
m_map);

                                update_local_symbol_table(s_info, &btarget_table);
                        }
                        morefiles =     move_to_next_file(&src_iterator);
                };

                /**update entries in the global symbol table;**/
                update_global_symbol_table(&symbol_table,  &btarget_table);
                /**update the bitfields in text section;**/
                text_section -> bitstring_offs = 0;
                size = comp_bitstring.first_unused_address - out_mod -> binary_image;
                /* If size is not a multiple of ICACHE_BLOCK_SIZE, then
                append padding bytes and set the padding bytes to 0.
                */
                pad_size = ICACHE_BLOCK_SIZE - (size % ICACHE_BLOCK_SIZE);
                for (pad_num=0; pad_num < pad_size;
                                pad_num++){
                        *(comp_bitstring.first_unused_address+pad_num) = 0;
                }
/*
                fprintf(stdout, "Size of text section before compression =");
                fprintf(stdout, "%lu (num. ins), %lu(num bytes @27 bytes/ins)\n",
                        text_section->size.lo, text_section->size.lo*27);
*/

                text_section -> bitstring_len = size * 8;
                size += pad_size;
/*
                fprintf(stdout, "Size of text section after  compression = %lu(num. byte
s)\n", size);
*/

                comp_bitstring.first_unused_address += pad_size;
/*              text_section -> bitstring_len = size * 8; */
#if 0
                Address_clear(text_section -> base);
#endif
                bit_vector_to_Address(&(text_section -> size), sizeof(byte *)*8,
                                        (byte *)&size);
                text_section -> mem_width = 8;
                text_section -> block_size = ICACHE_BLOCK_SIZE;
                base_address =  out_mod -> binary_image;
                update_bitfields(base_address, &ref_table, &btarget_table, out_mod);
                text_section -> ref_tbl_offs =
                        comp_bitstring.first_unused_address - base_address;
                base_address = comp_bitstring.first_unused_address;


/*------------------------------------------------------*/
/* Do the bit shuffling for the 256 bit blocks.                 */
/*------------------------------------------------------*/
                if (!G_all_globals.flags.shuffleoff)
                        bits_shuffle(out_mod, &ref_table, &comp_bitstring);
                base_address = reference_table_pack(base_address, &ref_table);
                reference_table_free(&ref_table);

/*------------------------------------------------------*/
        }
/*              Part III. update the bitfields in the
                                remaining sections.
                For each section that is not the text section
                        i) create the reference table;
                        ii)updtate the bitfields to reflect the new values
                                of the branch targets.
                        iii) copy the modified bitstring to the out_mod's
                        binary string.
                iv) pack the reference table in the out_mod's binary
                        image.
*/
/*-----------------------------------------------------------*/

for (section_id = 1; section_id <= inmod -> section_tbl_size;
                section_id++)
{
        section = (*(inmod -> section_tbl))[section_id-1];

/** 1.create_reference_table() for that section; **/

        if ((section_id != text_section_id) ){
                if (section -> has_bitstring){
                create_reference_table(inmod, &ref_table, section);

/* 2. step through the reference table and update
        bitfields when needed. We are updating in the
        input module itself as this is easier to handle.
        Actually this is not true : we might as well copy
        into the output module and then update. */

        section_start = inmod -> binary_image + section -> bitstring_offs;

        update_bitfields(section_start , &ref_table,   &btarget_table, out_mod);

/** 3 compute the position to start appending this section;
        append the section in that place; */

        section -> bitstring_offs = base_address - out_mod -> binary_image;
        size = section -> bitstring_len / 8;
        if (section -> bitstring_len % 8 !=0) size++;
        memcpy((void *)base_address, (void *)section_start, size);
        base_address = base_address + size;
        section -> ref_tbl_offs = base_address - out_mod -> binary_image;
        base_address = reference_table_pack(base_address, &ref_table);
        reference_table_free(&ref_table);

        } else{
                section -> bitstring_offs = base_address - out_mod -> binary_image;
                section -> ref_tbl_offs = base_address - out_mod -> binary_image;
                }
        }
}
binary_size = base_address - out_mod -> binary_image;
/*------------------------------------------------------------------*/
/*                      Part IV. Fill in the details for the
                                        global section.
                        */
/*------------------------------------------------------------------*/

/* -1. write linking history into a string and get history_size. */


        cal_time = time(NULL);
        local_time = localtime(&cal_time);
        if ( (timestamp = salloc(28)) == NULL )
                        STD_ERROR("Memory allocation failure.");
        strftime(timestamp, 29,
                                "%a %b %d %Y %H:%M:%S %Z", local_time);
        timestamp[28] = '\0';

        if (!TESTING){
                sprintf(history_string, "%s;%s;%s",timestamp, version_str, cmdstring);
                history_size = strlen(history_string) + 1;/*  +1 for '\0'. */
```

```
                history_size += inmod->history_size;
        }else{
                history_size = inmod->history_size;
                history_string[0] = '\0';
        }

/*      0. estimate the size of global partition and malloc. */

        global_partition_size = 0;
        global_partition_size += inmod -> string_tbl_size;
        global_partition_size +=
                (inmod -> section_tbl_size) * LIFE_Obj_sct_size;
        global_partition_size +=
                (inmod -> symbol_tbl_size) * LIFE_Obj_sym_size;
        global_partition_size += history_size;
        global_partition_size +=
                (inmod -> source_tbl_size) * LIFE_Obj_src_size;
        scatter_table_size =
                get_scatter_table_size(out_mod -> scatter_table);
        global_partition_size += scatter_table_size;
        global_partition = (byte *)malloc(global_partition_size);

/*      1. copy string table. */

        memcpy((void *)global_partition,
                (void *)(inmod -> memory_image + inmod -> string_tbl_offs),
                inmod -> string_tbl_size);

/*      2. pack section table. */

        mptr = global_partition + inmod -> string_tbl_size;
        for (i=1; i <= (int) (inmod -> section_tbl_size); i++){
                mptr = packSectionDescr((*inmod -> section_tbl)[i-1], mptr);
        }

/*      3. pack global symbol table. */

        symtab_next(&symbol_table, TRUE);
        while ( (symbol = symtab_next(&symbol_table, FALSE)) != NULL )
                mptr = packSymbolDescr(symbol, mptr);

/*      4. write linking history. */
        if (strlen(history_string) != 0){
                sprintf((char *)mptr, "%s", history_string);
                mptr += strlen(history_string) + 1; /* +1 for \0 .*/
        }
        memcpy((void *)mptr, (void *)(inmod -> global_image + inmod -> history_offs),
                                inmod -> history_size);
        mptr += inmod -> history_size;

/*      5. pack source table. */

        for (i = 0; i <(int) (inmod -> source_tbl_size); i++) {
                mptr = packSourceInfoDescr((*inmod -> source_tbl)[i], mptr);
    }

/*  6. pack scatter table. */
        mptr = scatter_table_pack(mptr, out_mod -> scatter_table);

/*-----------------------------------------------------------*/
/*                      Part V. create memory image of header.
*/
/*-----------------------------------------------------------*/
        out_mod -> magic =  inmod -> magic;
    out_mod -> version =  inmod -> version;
    out_mod -> version_str = inmod -> version_str;
    out_mod -> machine_str = inmod -> machine_str;
    out_mod -> machine_check = inmod ->  machine_check;
    out_mod -> flags = inmod -> flags | LIFE_Obj_is_compressed;
        if (inmod -> flags & LIFE_Obj_has_start){
        out_mod -> start_section = inmod -> start_section;
                if ((btarget = btarget_table_lookup(&btarget_table,
                                &inmod -> start_address)) == NULL){
                        STD_ERROR("Branch target lookup failure");
                }
        out_mod -> start_address = btarget -> new_address;
        }

    /* file partitioning */
    out_mod -> binary_offs = LIFE_Obj_header_size + global_partition_size;
    out_mod -> source_offs = out_mod -> binary_offs + binary_size;
        src_partition_size = compute_source_offsets(&inmod -> source_table);
    out_mod -> length = LIFE_Obj_header_size + global_partition_size
                        + binary_size +  src_partition_size;

/* global partition information */
    out_mod -> string_tbl_offs = inmod -> string_tbl_offs;
    out_mod -> string_tbl_size = inmod -> string_tbl_size;
    out_mod -> section_tbl_offs = inmod -> section_tbl_offs;
    out_mod -> section_tbl_size = inmod -> section_tbl_size;
    out_mod -> symbol_tbl_offs = inmod -> symbol_tbl_offs;
    out_mod -> symbol_tbl_size = inmod -> symbol_tbl_size;
    out_mod -> history_offs = inmod -> history_offs;
    out_mod -> history_size = history_size;
    out_mod -> source_tbl_offs = inmod -> history_offs + history_size;
    out_mod -> source_tbl_size = inmod -> source_tbl_size;
    out_mod -> scatter_tbl_offs = out_mod -> source_tbl_offs +
                                        out_mod -> source_tbl_size*LIFE_Obj_src_
size;
    out_mod -> scatter_tbl_size = scatter_table_size;


 /*------------------------------------------------------------*/
 /*          Part VI. write output onto file                              */
/*-------------------------------------------------------------*/

/*      1. open file */
        if ( (output_file = fopen(out_mod -> pathname, "wb")) == NULL ){
                sprintf(error_str,"Could not open file %s.",
                                out_mod -> pathname);
                printf("error str is %s\n", error_str);
                printf("pathname is %s\n", out_mod -> pathname);
                STD_ERROR(error_str);
        }
/*      2. write header */
        md_writeHeader(out_mod, output_file);

/*      3. write global partition.*/
        if ( fwrite(global_partition,  1, global_partition_size, output_file) != global_
partition_size ){
                sprintf(error_str, "Write failure while writing to %s.",
                                        out_mod -> pathname);
                STD_ERROR(error_str);
        }

/*      4. write binary partition. */
        if ( binary_size != 0 &&
                (fwrite(out_mod ->binary_image, 1, binary_size, output_file) != binary_s
ize )){
                sprintf(error_str, "Write failure while writing to %s.",
                                        out_mod -> pathname);
                STD_ERROR(error_str);
        }
/*      5. write src partition (directly from input module where
                it was updated. */
        size = inmod -> length - inmod -> source_offs;
        if ( size != 0 && fwrite(inmod -> source_image, 1, size, output_file) != size ){

                sprintf(error_str, "Write failure while writing to %s.",
                                        out_mod -> pathname);
                STD_ERROR(error_str);
        }

/* ---------------------------------------------------------*/
/*  Now close all the files and free all the memory used up. */
/*----------------------------------------------------------*/

        fclose(output_file);
        /* free all the data structures associated with the modules
                and then free the module structure itself. */
        free(inmod->memory_image);/* We have used md_readFile() and that
                        allocates memory for the entire memory image. */
        sct_destroy(inmod->section_tbl);
        scatter_table_free(inmod->scatter_table);
        srctab_destroy(&(inmod -> source_table));
        inum_map_destroy(inum_map);
        free(inmod);

        /* Do the same for output module. */
        free(out_mod->binary_image);
        free(global_partition);
        scatter_table_free(out_mod->scatter_table);
        free(out_mod);

        btarget_table_free(&btarget_table);
        symtab_free(&symbol_table);

}

main(int argc, char **argv)
{
char inmod_name[MAX_MODNAME_LENGTH +1],
        outmod_base[MAX_MODNAME_LENGTH +1],
        mapmod_name[MAX_MODNAME_LENGTH +1],
        fname[MAX_MODNAME_LENGTH +1];
int fcount;
Command *cmd;
int i;
NewSwitchDescr * sw;
char *help_str = " -v prints the version of the compressor.\n \
-h prints this information\n \
<file_name> input module name.\n";
char *usage_str = "Usage : tmcomp [-v] [-h] [-o=<output_file_name>] <file_name>\n";

#if 0
        fcount = 1;
        while (fcount <= argc-1){
                strcpy(inmod_name, argv[fcount++]);
                pname_to_fname(inmod_name, fname);
                basename(fname, outmod_base);

                main_compressor(inmod_name, outmod_base);
        }

#endif

        G_all_globals.flags.padoff = FALSE;
        G_all_globals.flags.shuffleoff = FALSE;
        G_all_globals.flags.mapon = FALSE;
        G_all_globals.flags.is_outmod_given = FALSE;
        cmd = parse_cmdline(argc, argv);
        for (i=0; i < num_switches(cmd); i++){
                sw = get_ith_switch(cmd, i);
                if (strcmp("v", get_switch_name(sw)) == 0){
                        fprintf(stderr, "%s", version_str);
                }
                else if (strcmp("h", get_switch_name(sw)) == 0){
                                fprintf(stderr, "%s\n%s", usage_str, help_str);
                                exit(0);
                }else if (strcmp("padoff", get_switch_name(sw)) == 0){
                                G_all_globals.flags.padoff = TRUE;
                }else if (strcmp("shuffleoff", get_switch_name(sw)) == 0){
                                G_all_globals.flags.shuffleoff = TRUE;
                }else if (strcmp("mapon", get_switch_name(sw)) == 0){
                                G_all_globals.flags.mapon = TRUE;
                }else if (strcmp("o", get_switch_name(sw)) == 0){
                                G_all_globals.flags.is_outmod_given = TRUE;
                                if (num_args_in_switch(sw) != 1){
                                        fprintf(stderr, "%s", usage_str);
                                        exit(1);
                                }
                                G_all_globals.out_mod_name = get_ith_arg_in_switch(sw, 0
);
                                if (strlen(G_all_globals.out_mod_name) >= MAX_MODNAME_LE
NGTH){
                                         fprintf(stderr, "output module name is too long
; Has to be less than %d chars\n", MAX_MODNAME_LENGTH);
                                        exit(1);
                                }
                }else{
                        fprintf(stderr, "%s", usage_str);
                        exit(1);
                }

        }

        for (i=0; i < num_nonswitch_inputs(cmd); i++){
                strcpy(inmod_name, get_ith_nonswitch_input(cmd, i));
                pname_to_fname(inmod_name, fname);
                basename(fname, outmod_base);
                basename(fname, mapmod_name);
                strcat(mapmod_name, ".map");

                main_compressor(inmod_name, outmod_base, mapmod_name);
        }
        exit(0);
}
```

```
/************************************************************/
/*        File: comp_src.c                                  */
/*        Author: Hari Hampapuram                           */
/************************************************************/
/* Associated files:
        moduleio.c is needed as pack and unpack functions for
        source descriptors are used.
        IS_TEXT_SECTION_ID() from comp_main.c is used which is very bad.
*/
/*      Functions defined in this file:
                1. begin_src_files_iterate(SourceFileIterator *src_iter,
                                SourceTable *src_tab);
                2. move_to_next_file(SourceFileIterator *src_iter);
                3. update_current_link_map(SourceFileIterator *src_iterator,
                   Address *begin_last_src_file,
                                Address *size_last_src_file);
                4.SourceInfoDescr *
                        get_current_src_descr(SourceFileIterator *src_iterator)
*/

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

/****** files from linker **********/
#include "types.h"
#include "lifetypes.h"
#include "lifeobj.h"
#include "scatter_types.h"
#include "salloc.h"
#include "stringtab.h"
#include "linktypes h"
#include "libtypes h"
#include "error.h"
#include "moduleio.h"
#include "libio.h"
#include "sectiontab.h"
#include "symboltab.h"
#include "symdump.h"
#include "symbolmap.h"
#include "sourcetab.h"
#include "mergeglobal.h"
#include "mergebinary.h"
#include "mergedebug.h"
#include "cmdline.h"


/****** files from compressor ********/
#include "compressor.h"
#include "comp_src.h"

/*-------------------begin_src_files_iterate()--------------------*/
/*
        The bitstring of the text section is made up of chunks of
        bitstrings, each from one source file. The bitstring is
        processed by compressing the chunks corresponding to one file
        at a time. The link map entries should have the source files
        in the proper order so that the entire code is processed from
        beginning to end continuously. (It is not assumed anywhere
        that this is the case, but if this is not the case, the
        ordering of the code w.r.t. th link map entries in the old
        nad new files could be different.)

        Reads the link map entries of the first file in the
        src file partition and sets up current_link_map of
        src_iter to  the section pointer of
        the text section or the last section of the file.

        NOTE: It is assumed that each file has at least one section.
                Also, each source file also needs a source table entry.

        This module should have made use of src_createAndLoad. But as it is
        it does not make use of it now. It so happens that now I do call
        src_createAndLoad in read_input_module() as this is needed for
        compute_source_offsets(). Thus code is kind of duplicated.
*/

void
begin_src_files_iterate(SourceFileIterator *src_iter,
                        SourceTable *src_tab)
{
int i;
byte *mem_addr;
SourceInfo *table;

        src_iter -> current = 0;
        src_iter -> src_table = src_tab;
        table = *(src_tab -> srcs);/* src_tab -> srcs is a pointer to
                an array of SourceInfo. Thus *(src_tab -> srcs) will be
                the actual array itself. */
        mem_addr = table[0] -> module -> source_image +
                        table[0] -> link_map_offs;

        /* may be clear current_link_map in src_iter */

        for (i=0; i < (int)(table[0] -> link_map_size); i++){
                mem_addr = unpackLinkMapEntryDescr(&(src_iter -> current_link_map),
                                                mem_addr);
                if (IS_TEXT_SECTION_ID(src_iter -> current_link_map.section,
                                        table[0] -> module))
                        break;
        }
}

/*-------------------move_to_next_file()---------------------------*/
/*
        Similar to the above function. This moves to teh next src file
        in the source table. As above this will move to the
        text section if it exists or to the last section of the src file
        if it doesn't exist. returns 1 if there was a file in the
        src table that was not iterated yet and 0 otherwise.
*/

move_to_next_file(SourceFileIterator *src_iter)
{
int current_idx;
byte *mem_addr;
int i;
SourceTable *src_tab;
SourceInfo *table;

        current_idx = ++(src_iter -> current);
        src_tab = src_iter -> src_table;
        table = *(src_tab -> srcs);
        if ( current_idx < (int)(src_tab -> occupancy)){
                mem_addr = table[current_idx] -> module -> source_image +
                                table[current_idx] -> link_map_offs;
                for (i=0; i <(int)(table[current_idx] -> link_map_size); i++){
                        mem_addr = unpackLinkMapEntryDescr(&(src_iter -> current_link_ma
p),
                                                        mem_addr);
                        if (IS_TEXT_SECTION_ID(src_iter -> current_link_map.section,
```

© 1995 Philips Electronic North America Corporation

```
                                    table(current_idx) -> module))
                        break;
                }
                return (1); /*indicating more files */
        }
        return(0); /*no more files*/
}

/*--------------------update_current_link_map()-----------------*/
/* change the values of the current_link_map to the values
        given by the arguments. Needed for updating the values
        of the link map table when the code is being compressed.
*/

void
update_current_link_map(SourceFileIterator *src_iterator,
                        Address *begin_last_src_file,
                                        Address *size_last_src_file)
{
SourceTable *src_table;
int current_idx;
byte *mem_addr;
int i;
SourceInfo *table;
LinkMapEntryDescr temp_link_map;

        src_table = src_iterator -> src_table;
        table = *(src_table -> srcs);
        current_idx = src_iterator -> current;
        mem_addr = table[current_idx] -> module -> source_image
                                + table[current_idx] -> link_map_offs;
        for (i=0; i < (int)(table[current_idx] -> link_map_size); i++){
                unpackLinkMapEntryDescr(&temp_link_map, mem_addr);
                if (temp_link_map.section == src_iterator -> current_link_map.section){
                        temp_link_map.base =  *begin_last_src_file;
                        temp_link_map.size =  *size_last_src_file;
                };
                mem_addr = packLinkMapEntryDescr(&temp_link_map, mem_addr);
        }
}

/*----------------------get_current_src_descr()--------------------*/
/*
        returns the current src descriptor.
*/

SourceInfoDescr *
get_current_src_descr(SourceFileIterator *src_iterator)
{
SourceTable *src_table;
SourceInfo *table;

        src_table = src_iterator -> src_table;
        table = *(src_table -> srcs);
        return(table[src_iterator -> current]);
}
```

```
/**************************************************************/
/*                 File:   comp_reference.c
                   */
/*          Author: Hari Hampapuram                            */
/*************************************************************/
/* Associated files:
        Routines from moduleio.c (unpack and packRefDescr() are
        called.
*/
/* Functions defined in this file
  1. unsigned long
                reference_table_init(ReferenceTable *ref_table,
                                                             unsigned long tbl_size)

  2. RefDescr *get_new_ref_descr()
  3. reference_table_add(ReferenceTable *ref_table, RefDescr *ref)
  4. int ref_descr_compare(RefDescr **in_1, RefDescr **in_2)
     This is static.
  5. RefDescr *
                get_current_reference(ReferenceTable *ref_table)
  6. reference_table_next(ReferenceTable *ref_table, boolean reset)
  7. create_reference_table(Module inmod, ReferenceTable *ref_table,
                                          Section input_section)
  8. byte *
                reference_table_pack(byte *mem_addr, ReferenceTable *ref_table)
  9. void
                reference_table_free(ReferenceTable *ref_table)
*/

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

/******* files from linker ***********/
#include "types.h"
#include "lifetypes.h"
#include "lifeobj.h"
#include "scatter_types.h"
#include "salloc.h"
#include "stringtab.h"
#include "linktypes.h"
#include "libtypes.h"
#include "error.h"
#include "moduleio.h"
#include "libio.h"
#include "sectiontab.h"
#include "symboltab.h"
#include "symdump.h"
#include "symbolmap.h"
#include "sourcetab.h"
#include "mergeglobal.h"
#include "mergebinary.h"
#include "mergedebug.h"
#include "cmdline.h"

/******* files from compressor *********/
#include "compressor.h"
#include "comp_reference.h"

/*------------------1.reference_table_init()------------------------*/
/*
        1. ref_table  - must point to a struct for which space has been
                                allocated. This routine allocates space for
                                the actual table within the struct.
        2. tbl_size   - number of reference descriptors that the table can
                                hold.
```

```
*/

unsigned long
reference_table_init(ReferenceTable *ref_table, unsigned long tbl_size)

{
        if ((ref_table -> table =
                (RefDescr **) malloc((tbl_size+1) * sizeof(RefDescr *))) == NULL){
                MALLOC_ERROR;
        }

        ref_table -> num_descriptors = 0;
        ref_table -> capacity = tbl_size;
        ref_table -> sorted = FALSE;
        return(tbl_size);
}

/*----------------------get_new_ref_descr()----------------------*/
/* mallocs space for a reference descriptor and returns a pointer to
        it.
*/

RefDescr *get_new_ref_descr()
{
RefDescr *ptr;

        if ((ptr = (RefDescr *)malloc(sizeof(RefDescr))) == NULL){
                MALLOC_ERROR;
                exit(1);
        }
        return(ptr);
}

/*-------------------reference_table_add()------------------------*/
/*
        1. ref_table  - must have been initialized by a call to
                                reference_table_init(). There must be space
                                in it to hold a new descriptor. This does not
                                reallocate space if the capacity is exceeded.
        2. ref        - must be a valid descirptor address. If not
                                the reference table will have a junk value.
*/

reference_table_add(ReferenceTable *ref_table, RefDescr *ref)
{
        if (ref_table -> num_descriptors == ref_table -> capacity ){
                LOG_ERROR("Reference table capacity exceeded.");
        }
        ref_table->table[ref_table -> num_descriptors++] = ref;
        ref_table -> sorted = FALSE;
        return(ref_table -> num_descriptors);
}

/*--------------------ref_descr_compare()--------------------*/
/*
        For using the qsort to sort the reference descriptor table,
        we need this function. in_1 <= in_2 if in_1's position  <=
        in_2's position in the bitstring.
*/

static int
ref_descr_compare(const void *in_1, const void *in_2)
/*ref_descr_compare(RefDescr **in_1, RefDescr **in_2)*/
{
        return((*(RefDescr **)in_1)->position - (*(RefDescr **)in_2)->position);
}
```

```
/*--------------------get_current_reference()--------------------*/
/*
        If during the iteration of the reference table, the end has been
        reached, NULL is returned. The current descriptor is returned
        otherwise.
*/

RefDescr *
get_current_reference(ReferenceTable *ref_table)
{
        if (ref_table -> current >= ref_table -> num_descriptors)
            return NULL;
        return(ref_table -> table[ref_table -> current]);
}

/*--------------------reference_table_next()--------------------*/
/*
        1. ref_table - must be a properly initialized ref_table
                                containing valid data.
        2. reset     - if this is TRUE, a new iteration starts, NULL
                                is returned and ref_table -> current is set to 0

                                Also, the table is sorted if it has not been sor
ted
                                before or something has been added after sorting

                                if FALSE, it is assumed that the iteration has
                                been initialized by calling this routine with
                                reset == TRUE. ptr to next entry is returned if
                                there is a next entry, otherwise NULL is returne
d.
                                ref_table -> current is incremented.
*/

RefDescr *
reference_table_next(ReferenceTable *ref_table, boolean reset)

{
        if (reset == TRUE){
                ref_table -> current = 0;
                if (ref_table -> sorted == FALSE){
                        qsort(ref_table -> table, ref_table -> num_descriptors,
                                sizeof(RefDescr *), ref_descr_compare);
                        ref_table -> sorted = TRUE;
                }
                return(NULL);
        } else {
                if (ref_table -> sorted == FALSE){
                        LOG_ERROR("Unsorted reference table encountered.");
                }
        }


        if (ref_table -> current >= ref_table -> num_descriptors)
                return NULL;

        return(ref_table -> table[ref_table -> current++]);
}

/*---------------  ----create_reference_table--------------------*/
/*
        1. inmod  - The reference table entries of this module for the
                        given section (by input_section) are used to
                        create the entries of the ref_table. inmod's
                        offset's and size's must have been initialized and
```

```
                                so also the binary images.
        2. ref_table - must point to a valid struct (can not be NULL)
                                Space for this will get allocated depending of t
he
                                inmod's entries. This will have the reference
                                table of inmod in a usable form after this call.
i.e
                                we can call the various functions in this file.
        3. input_section - must be a valid ptr to a section desrciptor.
                                        A section of inmod.
*/

create_reference_table(Module inmod, ReferenceTable *ref_table,
                                        Section input_section)
{
byte *ref_address = inmod -> binary_image +
                                        input_section -> ref_tbl_offs;
unsigned long ref_count = input_section -> ref_tbl_size;
RefDescr *ref_ptr;

        reference_table_init(ref_table, ref_count);
        ref_table -> bitstring_base = inmod -> binary_image +
                                        input_section -> bitstring_offs;
        ref_table -> defmod = inmod;
        while (ref_count-- != 0){
                if ((ref_ptr = get_new_ref_descr()) == NULL){
                        LOG_ERROR("Could not create reference descriptor.");
                }
                ref_address = unpackRefDescr(ref_ptr, ref_address);
                reference_table_add(ref_table, ref_ptr);
        };
}

/*--------------------reference_table_pack()--------------------*/
/*
        1. mem_addr - The ref_table will be packed as it appears in the
                        object module, starting at mem_addr. There must be
                        sufficient space for the reference table.
        2. ref_table - must be having consistent entries.

        The function retruns the pointer to the byte immediately
        after the ref_table,f rom where one can start appending the
        next portion of the object module.
*/

byte *
reference_table_pack(byte *mem_addr, ReferenceTable *ref_table)
{
unsigned long ref_count = 0;
byte *current_address = mem_addr;

        while (ref_count < ref_table -> num_descriptors){
                current_address =
                        packRefDescr(ref_table -> table[ref_count], current_address);
                ref_count++;
        };
        return(current_address);
}

/*----------------------reference_table_free()---------------*/
/*
        Frees all the space malloc'ed for the reference table.
*/

void
reference_table_free(ReferenceTable *ref_table)
```

```
{
unsigned long ref_count = 0;

        while (ref_count < ref_table -> num_descriptors){
                free(ref_table -> table[ref_count]);
                ref_count++;
        }
        free(ref_table-> table);
}
```

```
/*****************************************************************/
/*         File: comp_misc.c                                     */
/*         Author: Hari Hampapuram                               */
/*****************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

/******* files from linker ***********/

#include "types.h"
#include "lifetypes.h"
#include "lifeobj.h"
#include "scatter_types.h"
#include "salloc.h"
#include "stringtab.h"
#include "linktypes.h"
#include "libtypes.h"
#include "moduleio.h"
#include "libio.h"
#include "sectiontab.h"
#include "symboltab.h"
#include "symdump.h"
#include "symbolmap.h"
#include "sourcetab.h"
#include "mergeglobal.h"
#include "mergebinary.h"
#include "mergedebug.h"
#include "cmdline.h"

/******* files from compressor *********/
#include "compressor.h"
#include "comp_misc.h"
#include "dump_structs.h"

/*****************************************************************/
/***              Miscellaniuos                           *******/
/*****************************************************************/
void cleanup(void)
{
        fprintf(stderr, "entered cleanup()\n");
}
void terminate(exit_status status)
{
        exit(status);
}


Section
GET_SECTION(Module md, char *sct_name)
{
int i;
Section sct;
        for (i=0; i < (int)(md -> section_tbl_size); i++){
                sct = (*md->section_tbl)[i];
                if (strcmp((sct -> name + md->string_tbl), sct_name) == 0){
                        return(sct);
                }
        }
        return(NULL);
}

SectionId
GET_SECTION_ID(Module md, char *sct_name)
{
int i;
Section sct;

        for (i=0; i < (int)(md -> section_tbl_size); i++){
                sct = (*md->section_tbl)[i];
                if (strcmp((sct -> name + md->string_tbl), sct_name) == 0){
                        return(i+1);
                }
        }
        return(-1);
}


IS_TEXT_SECTION(Section sct, Module md)
{

        if (strcmp((sct -> name + md->string_tbl), "text")==0){
                return(1);
        }
        return(0);
}

IS_TEXT_SECTION_ID(SectionId sct_id, Module md)
{
Section sct;

        if (sct_id == 0){
                return(0);
        };
        sct = (*md->section_tbl)[sct_id - 1];
        if (strcmp((sct -> name + md->string_tbl), "text")==0){
                return(1);
        }
        return(0);
}

void
pname_to_fname(char *pname, char *fname)
{
char *ptr;

     if ((ptr = strrchr(pname, '/')) == NULL){
          strcpy(fname, pname);
     } else {
          strcpy(fname, ptr+1);
     }
}

void basename(char *fname, char *bname)
{
char *ptr, *ptr2;
int i;

     if ((ptr = strrchr(fname, '.')) == NULL){
          strcpy(bname, fname);
     } else {
          for (ptr2 = fname, i=0; ptr2 != ptr; ptr2++, i++){
               bname[i] = *ptr2;
          }
          bname[i] = '\0';
     }
}
```

© 1995 ~~North~~ Philips Electronics North America Corporation

```
/*************************************************************/
/*          File: comp_btarget.c                              */
/*          Author: Hari Hampapuram                          */
/*************************************************************/

/* associated files:
        comp_reference.c is needed as reference tables are to be
        created for collect_btargets.
        comp_utils.c and comp_scatter.c are required for getting the
        bitfields to a proper value.
*/
/*
 Functions defined in this file:
  1. btarget_table_init(unsigned long tbl_size, BtargetTable *btarget_table);
  2. btarget_table_add(BtargetTable *btarget_table, TargetDescr *btarget);
  3. btarget_table_lookup(BtargetTable *btarget_table, Address instruction_address)
  4. btarget_table_update(BtargetTable *btarget_table, Address old_address,
  Address new_address)
  5. BtargetDescr *get_new_btarget()
  6. void collect_branch_targets(SymbolTable *sym_tab,
                                        BtargetTable *btarget_table, Module mod);
  7. void btarget_table_free(BtargetTable *btarget_table)
*/

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

/****** files from linker **********/
#include "types.h"
#include "lifetypes.h"
#include "lifeobj.h"
#include "scatter_types.h"
#include "salloc.h"
#include "stringtab.h"
#include "linktypes.h"
#include "libtypes.h"
#include "error.h"
#include "moduleio.h"
#include "libio.h"
#include "sectiontab.h"
#include "symboltab.h"
#include "symdump.h"
#include "symbolmap.h"
#include "sourcetab.h"
#include "mergeglobal.h"
#include "mergebinary.h"
#include "mergedebug.h"
#include "cmdline.h"

/****** files from compressor ********/
#include "compressor.h"
#include "comp_btarget.h"
#include "comp_reference.h"
#include "comp_scatter.h"
#include "comp_utils.h"
#include "comp_src.h"

static int use_sorted = 0;/* variable to indicate when to sort the
                                                btarget table. Using sort option
 while
                                                collecting the btargets seems wa
steful,
                                                so after collecting the btargets
, we turn
                                                this on. All the searches for br
anch targets
                                                                are on a sorted table. */
/*----------------------btarget_table_init()-------------------*/
/*
        1. tbl_size - number of btargets that the table is expected
                to hold. (this could be any positive number to begin
                with, but a good approximation will make the btarget table
                more efficient.
        2. btarget_table - must point to a BtargetTable structure.
*/

btarget_table_init(unsigned long tbl_size, BtargetTable *btarget_table)
{
        if ((btarget_table -> table =
                (BtargetDescr **)malloc((tbl_size )*sizeof(BtargetDescr *))) == NULL)(
                MALLOC_ERROR;
        );

        btarget_table -> num_targets = 0;
        btarget_table -> capacity = tbl_size;
        btarget_table -> sorted  = FALSE;
        return (tbl_size);
}

/*----------------------btarget_table_add()----------------------*/
/*
        adds btarget to btarget_table. If the table is currently
        full, more space is allocated and all the old pointers are copied.
        This function assumes that the target passed is not in the
        btarget table. Hence it does not do a lookup.
*/

btarget_table_add(BtargetTable *btarget_table,
                                                        BtargetDescr *btarget)

{

BtargetDescr **ptr;
unsigned long i;

        if (btarget_table -> num_targets == btarget_table -> capacity)
        {
        /*      if ((btarget_table ->  table =
                                realloc(btarget_table ->  table, 2*btarget_table -> capa
city)) == NULL)(
                        MALLOC_ERROR;
        }
        */
                if ((ptr =
                        (BtargetDescr **)malloc(2*(btarget_table -> capacity)*sizeof(Bta
rgetDescr *))) == NULL)(
                        MALLOC_ERROR;
                );
        /*      memcpy(ptr, btarget_table ->  table, btarget_table -> capacity*sizeof(Bt
argetDescr *));*/
                for (i=0; i < btarget_table -> num_targets; i++)(
                        ptr[i] = btarget_table -> table[i];
                )
                free(btarget_table ->  table);
                btarget_table ->  table = ptr;

                btarget_table -> capacity = 2*btarget_table -> capacity;
        )
        btarget_table -> table[ btarget_table -> num_targets++] = btarget;
        btarget_table -> sorted = FALSE;
        return(btarget_table -> num_targets);
}
```

```
/*--------------------btarget_compare()---------------------------*/
/*
For using the qsort to sort the btarget_table and for bsearch.
*/
static int
btarget_compare(const void *in_1, const void *in_2)
/*btarget_compare(BtargetDescr **in_1, BtargetDescr **in_2)*/
{
        return(Address_cmp((*(BtargetDescr **)in_1)->old_address,
                        (*(BtargetDescr **)in_2)->old_address));
}

BtargetDescr *
k_n_r_bin_search(BtargetTable *btarget_table, BtargetDescr *key)
{
        int cond;
        BtargetDescr **low = &(btarget_table->table[0]);
        BtargetDescr **high;
        BtargetDescr **mid;

        if (btarget_table->num_targets == 0) return NULL;
        high = &(btarget_table->table[btarget_table->num_targets-1]);
        while (low < high) {
                mid = low + ((high -low) / 2);
                if ((cond = btarget_compare(&key, mid)) < 0)
                        high = mid;
                else if (cond > 0)
                        low = mid + 1;
                else return *mid;
        }
        if (low == high){
                if (btarget_compare(&key, low) == 0)
                        return *low;
                else
                        return NULL;
        }
        return NULL;
}

static void
btarget_table_sort(BtargetTable *btarget_table)
{
        qsort(btarget_table -> table, btarget_table -> num_targets,
                        sizeof(BtargetDescr *), btarget_compare);
        btarget_table -> sorted = TRUE;
}
/*---------------------btarget_table_lookup()---------------------*/
/*
        Searches the btarget_table to see if there is an entry
        for instruction_address in it. If there is, the corresponding
        pointer is returned. A NULL is returned otherwise.
*/

BtargetDescr *
btarget_table_lookup(BtargetTable *btarget_table, Address *instruction_address)
{
unsigned long i;
BtargetDescr *btarget, *tmp_bt;
BtargetDescr dummy, *dummy_ptr;
/*go thro' the list and return the corresponding pointer or
 NULL;*/
if (use_sorted) {
        if (btarget_table -> sorted == FALSE){
                        btarget_table_sort(btarget_table);
        };
        dummy.old_address = *instruction_address;
        dummy_ptr = &dummy;
#if 0
        btarget = bsearch(&dummy_ptr, btarget_table->table,
                                        btarget_table -> num_targets, sizeof(BtargetDesc
r *),btarget_compare);
#endif
        btarget = k_n_r_bin_search(btarget_table, dummy_ptr);
#if 0
        tmp_bt = NULL;
        for (i = 0; i < btarget_table -> num_targets; i++){
                if (Address_cmp(*instruction_address,
                                btarget_table ->table[i] -> old_address) == 0){
                                tmp_bt = btarget_table -> table[i];
                }
        }
        if (tmp_bt != btarget){
                printf("oops!\n");
                dump_BtargetTable(btarget_table);
                exit(1);
        }
#endif
        return(btarget);
}else{
                for (i = 0; i < btarget_table -> num_targets; i++){
                        if (Address_cmp(*instruction_address,
                                        btarget_table ->table[i] -> old_address) == 0){
                                return (btarget_table -> table[i]);
                        }
                }
                return(NULL);
        }
}

/*------------------------btarget_table_update()------------------*/
/*
        The entry in btarget_table that has btarget_table->old_address ==
        old_address is updated so that btarget_table->new_address ==
        new_address. If old_address is not currently in btarget_table
        NULL is returned and otherwise, ptr to the updated btarget is
        returned.
*/

BtargetDescr *
btarget_table_update(BtargetTable *btarget_table,
                                Address *old_address, Address *new_address)
{
BtargetDescr * target;

        if ((target =
                        btarget_table_lookup(btarget_table, old_address)) == NULL)
                {
                        return(NULL);
                }
        target -> new_address = *new_address;
        return (target);
}

/*-------------------------get_new_btarget()----------------------*/

BtargetDescr *ge·_new_btarget()
{
BtargetDescr *ptr;
        if ((ptr = (BtargetDescr *)malloc(sizeof(BtargetDescr))) == NULL){
                MALLOC_ERROR;
        }
        return(ptr);
}

/*----------------------collect_branch_targets()------------------*/
/*
        sym_tab - is a pointer to the global symbol table of the module mod.
                Actually, it would have been better to have the mod initialized
                so that it had the sym_tab within it. In that case sym_tab
                is redundant. (I guess even now, mod has the syn_tab in
                it already.)
        btarget_table - will finally have all the branch targets that are
                defined in the module mod. These are all the entries in the
                global symbol table that are defined in the text section
                of this module, and all the entries in the reference tables
                of the sections in the binary partition that have a definition
                in the text section of this section.
        mod - this is needed for accessing the reference tables of the
                sections for collecting branch targets.
*/

void
collect_branch_targets(SymbolTable *sym_tab,
                                        BtargetTable *btarget_table, Module mod)

{
Symbol current_symbol;
RefDescr *current_ref;
Address tmp_address;
BtargetDescr *btarget;
byte  address_buffer[MAX_BYTES_PER_ADDPESS];
byte *bitstring_base, *byte_ptr;
ScatterDescr *s_descr;
int offset;
ReferenceTable ref_table;
SectionId section_id;
Section section;
unsigned long approx_size = 100;
SourceFileIterator src_iterator;
byte                *lptr;
SymDescr sym;
StringId            cnt;
int morefiles;
SourceInfoDescr *s_info;

        use_sorted = 0;/* turn off sorting while collecting branch targets.
                                        Sorting during the collection can lead to many
                                        calls to the sort routine and thus will be
                                        inefficient. (we are trading off search time
                                        with sorting time at this point.) We turn on
                                        sorting after collecting the branch targets, as
                                        we will have only searching at that point and no

                                        calls will be made to the sort routine.*/
        btarget_table_init(approx_size,  btarget_table);

/* Collect btargets from the global symbol table */
        symtab_next(sym_tab,TRUE);
        while ((current_symbol = symtab_next(sym_tab, FALSE)) != NULL){
                if (IS_TEXT_SECTION_ID(current_symbol -> section, mod)){
                        btarget = get_new_btarget();
                        btarget -> old_address = current_symbol -> value;
                        Address_clear(btarget -> new_address) ;
                        if (btarget_table_add(btarget_table, btarget) < 0){
                                LOG_ERROR("Could not add branch target to table.");

                        }
```

```
                }
        }

/* Collect btargets from the local symbol tables */
        begin_src_files_iterate(&src_iterator, &(mod -> source_table));
        morefiles = 1;
        while (morefiles){
                s_info = get_current_src_descr(&src_iterator);
                lptr = s_info->module->source_image + s_info->sym_tbl_offs;
                for (cnt = s_info->sym_tbl_size; cnt != 0; --cnt) {
                        unpackSymbolDescr(&sym, lptr);
                        if (IS_TEXT_SECTION_ID(sym.section, s_info->module)) {
                                if ((btarget =
                                        btarget_table_lookup(btarget_table, &(sym.value)
))==NULL){
                                                btarget = get_new_btarget();
                                                btarget -> old_address = sym.value;
                                                Address_clear(btarget -> new_address) ;
                                                if (btarget_table_add(btarget_table, bta
rget) < 0){
                                                        LOG_ERROR("Could not add branch
target to table.");
                                                }
                                        }
                                }
                        lptr = packSymbolDescr(&sym, lptr);
                }
        morefiles = move_to_next_file(&src_iterator);
        }
/* Get the btargets from teh reference tables. */

        for (section_id = 1; section_id <= mod -> section_tbl_size;
                        section_id++)
        {
                section = (*(mod -> section_tbl))[section_id-1];
                create_reference_table(mod, &ref_table, section);
                reference_table_next(&ref_table, TRUE);
                bitstring_base = ref_table.bitstring_base;
                while ((current_ref = reference_table_next(&ref_table, FALSE)) != NULL){

                        if (IS_TEXT_SECTION_ID(current_ref -> section, mod)){
                                byte_ptr = bitstring_base + (current_ref -> position / 8
);
                                offset = current_ref -> position % 8;
                                s_descr = scatter_descr_from_id(current_ref -> scatter_t
ype, mod -> scatter_table);
                                extract_and_arrange_bits(byte_ptr, offset,  s_descr, add
ress_buffer);
                                bit_vector_to_Address(&tmp_address, s_descr -> total_wid
th,
                                                address_buffer);
                                if ((btarget_table_lookup(btarget_table, &tmp_address))

                                        == NULL) {
                                        btarget = get_new_btarget();
                                        btarget -> old_address = tmp_address;
                                        Address_clear(btarget -> new_address) ;
                                        if (btarget_table_add(btarget_table, btarget) <
0){
                                                LOG_ERROR("Could not add branch target t
o table.");
                                        }
                                }
                        }
                };
                reference_table_free(&ref_table);
        )
        use_sorted = 1;

)

void
btarget_table_free(BtargetTable *btarget_table)
(
int i;

        for (i=0; i < btarget_table -> num_targets; i++)(
                free(btarget_table -> table[i]);
        )

        free(btarget_table);
)
```